

How to calculate 3D coordinates with two cameras, a calibration object, a java program, and a lot of MS Excel macros

Lon Kelly
lkelly@ak.blm.gov

June 10, 2002

1 Warnings

The Java program and Excel Macros described here are not nearly industrial strength. They don't check their inputs. They generally don't give warnings. They will give bad results if you aren't careful. Experiment until you know things are working as you expect.

You need to be able to digitize points on individual video frames to implement this method. I don't plan to develop a digitizing program for Macintosh or Windows until Java provides platform independent classes that will provide video-in-a-window and camera/recorder control for single stepping through frames.

This work generalizes a series of programs¹ I wrote to develop 3D representations of fish mid-lines after they were video taped swimming around in a stream. The earlier work was done on a NeXTDIMENSION computer, and provided many enhancements and a decent user interface.

The work described below has no user interface and no enhancements. It only deals with single points—not midlines. I used Java and Excel/Visual Basic because they are multi-platform (*i.e.* Mac and Windows) and flexible. Based on several requests, I wanted to have some way for people to get started or process a small amount of data without taking on a significant programming task. This is really just an implementation of the core concepts of our method, generalizing the 3D geometry. If I were using this, I would definitely customize it for my experimental setup.

The code has only had rudimentary testing, and probably has bugs. If you use it, it's at your own risk.

2 Concepts

In this section, I describe the steps required to pursue a generalized brute force method to take a set of observations made with at least two cameras and resolve them into 3D coordinates. The method more fully described by Hughes and Kelly [2]. It should be noted that there is a typographical error on page 2476 of [2]: $i = h^2 - k^2 - l^2$ should be $i = h^2 + k^2 + l^2$. The math behind the coordinate transformations is described

¹These programs are available at on request to: lkelly@ak.blm.gov

in computer graphics texts, such as Hearn and Baker [1]. The math behind the least-squares calculations is described briefly by Press *et al* [3].

2.1 Reference object

Using our method, the experimenter will place two (or more) cameras so they have a view of the area where the experiment will be observed. She will then place a reference object of known dimensions so it is visible in both cameras. Once the reference object has been recorded, it can be removed until the cameras are moved. The reference object will generally be a transparent or mesh cube or rectangular prism (a quadrat), and will generally be oriented so each camera has a view of two faces—one *near* face, and one *far* face. The reference object provides a 3D grid of known points, allowing us to adjust for many kinds of optical distortions.

2.2 Coordinate systems

We use three coordinate systems: *view*, *face*, and *world*.

2.2.1 The view coordinate system

The view coordinate system is two dimensional and cartesian. Points in view coordinates represent a location on your computer screen within the screen area used to display the images you are digitizing. For example, the lower left corner of your video view might have view coordinate (0,0), and the upper right might be (640,480). The units are probably screen pixels.

2.2.2 The face coordinate system

Each face of the reference object will be marked to define at least 10 points. These points are expressed in what I call *face* coordinates. For example, you might call the marked point at the lower left of the face, as viewed by the camera, (0,0). There is one face coordinate system defined by each face of the quadrat. The face coordinate systems are two dimensional and cartesian. They are used to determine the parameters of polynomials that can convert between the 2D view coordinate system and the face coordinate system. The units are probably centimeters.

2.2.3 The world coordinate system

The world coordinate system is three dimensional. It represents the coordinate system you want to use when you are analyzing the 3D data from your experiments. The units are probably centimeters.

3 Transforming coordinates

3.1 View coordinate system to face coordinate system

Each view coordinate tuple is equivalent to a ray of light traveling through the environment to strike the detector in your camera. We know the ray of light followed a curved path when it went through the air/water interface and the camera optics. We know our

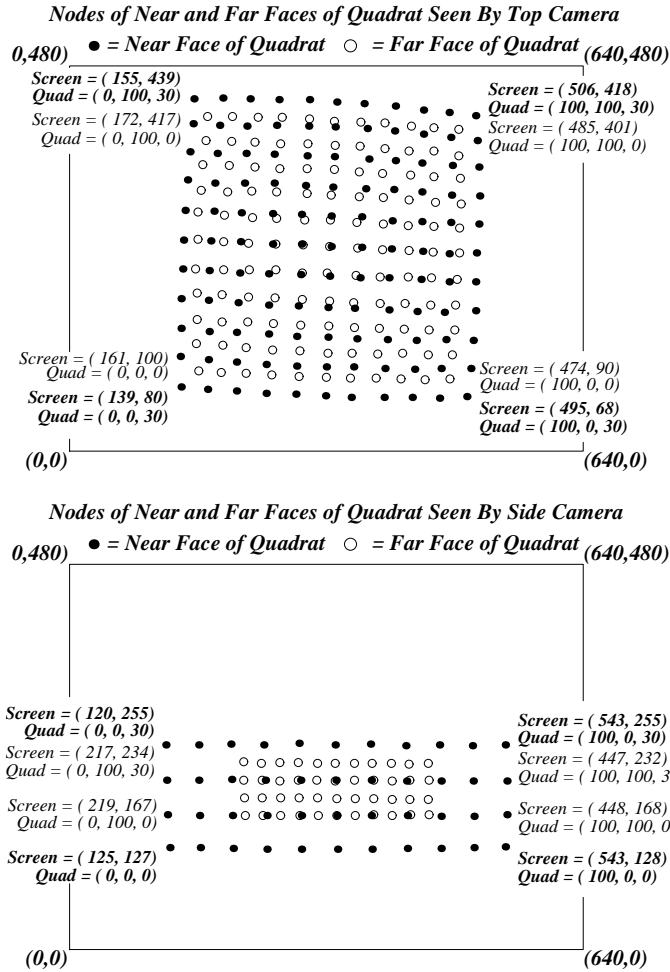


Figure 1: Plots of the quadrat nodes as seen in the digitizing view.

rectangular quadrat faces appear skewed and rounded on the screen. See Figure 1² for a look at this situation. Our first job is to remove these effects, to the extent possible. We do this by fitting the parameters of two polynomial equations that can convert between screen coordinates and face coordinates. This method was adapted from the GRASS Geographic Information System rectification module. If we have two cameras with two face coordinate systems each, then we have four conversions to do, which requires eight equations.

I use a Java command line program to calculate the parameters for each face coordinate system. The input to the program consists of pairs of points. The first point in the pair is the screen coordinates observed for one of the quadrat nodes, and the second point of the pair is the face coordinates of the same node.

The Java class that does the calculations is named “Hello” for historical reasons, and will change. To run it, type `java Hello infile.txt outfile.txt` where

²Nick Hughes at the University of Alaska, Fairbanks, made the figures for this paper.

`infile.txt` is the input file, and `outfile.txt` is the output file. The real work of the program is done by the Jama matrix math package³, so both `Hello.class` and the Jama directory must be in your `CLASSPATH`. Jama uses QR decomposition to solve the least squares problems defined by the matrices constructed by `Hello`

Once you have the parameters, the actual conversion to face coordinates can be done in Microsoft Excel, using the macro `rectify(xP, yP, x, y)`, which is written in Visual Basic, and somewhat documented in the code. This macro returns an array representing a 2D point. Each view coordinate pair can then yield 2 face coordinate pairs, one for the near face, and one for the far face for the camera.

3.2 Face coordinate system to world coordinate system

At this point we need to convert the 2D face coordinates to 3D world coordinates. To do this, we construct a 4 by 4 *coordinate transformation matrix* as described by Hearn and Baker [1]. We build the CTMs by Excel macros, written in Visual Basic, and described in the code.

Using the macro `transformPoint(ctm, point)` we multiply the each CTM by the corresponding face coordinates (which we convert to 3D in the form (faceX, faceY, 0)) to yield 3D world coordinates. Note that `transformPoint()` uses *homogeneous* coordinate tuples internally, and actually returns its result as a homogeneous form as a four element array $(x, y, z, 1)$.

4 The 3D line intersection problem

After going through all the coordinate transformations, we have converted each (x, y) tuple we digitized in the view coordinate system into two (x, y, z) tuples in the world coordinate system. These pairs of (x, y, z) tuples define lines of sight through 3D space. The intersection of lines of sight for the same object at the same time, as observed by two cameras, will be the 3D location of the object we're interested in.

The calculations for this step can be done by the Excel macro `threeDLineIntersection(p1, p2, p3, p4)` which takes four 3D points as input and returns a single 3D point as the intersection of the two lines defined by the points. This macro is somewhat documented in the code. The local array variable `distance` is filled with a 3D vector representing the distance between the two lines at their closest points. The length of this vector indicates the error of the observation. This could be returned as well as the "intersection."

5 Worked example

This section refers to an Excel workbook, **TrakBookExample.xls**, distributed with this documentation. It is made up of several related worksheets and a module containing the macros written in Visual Basic.

5.1 Setup

This setup differs from the one used to generate Figure 1—the quadrat is the same, but it is in a different position in view coordinates in the data set for this worked example.

³Thanks to The Math Works, Inc., and the National Institute of Standards.

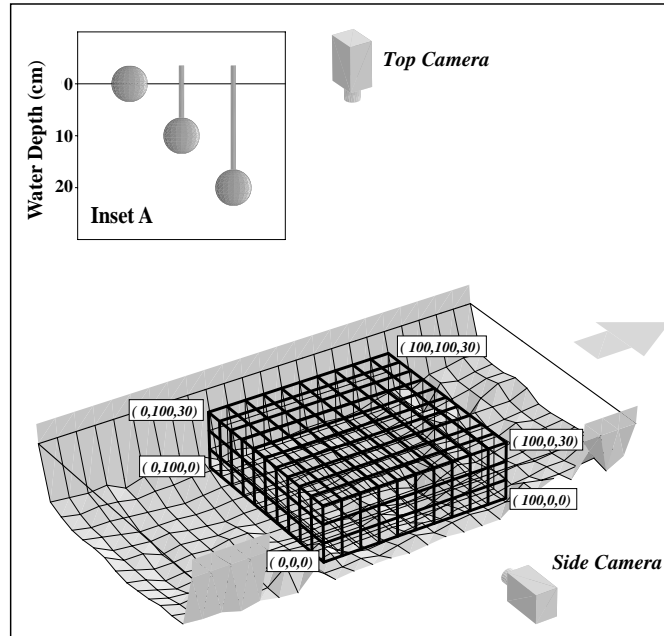


Figure 2: Field setup

In this example we will look at some test data that are easy to visualize and follow through the calculations. The test data are the 8 corners of the quadrat. The view coordinates of these corners are the first 8 points entered on the **ViewData** worksheet.

Figure 2 shows the field setup. The world coordinate system has the positive x-axis pointing downstream, the positive y-axis pointing at the left bank, and the positive z-axis pointing up. The origin of the world coordinate system is defined by a corner of the quadrat. One underwater camera (the side camera) looks across the stream so the origin is near the lower left of the video image, the positive x-axis points to the right, and the z-axis points up. The other camera (the top camera) is suspended above the stream, the origin is near the lower left of the video image, the positive y-axis points up, and the positive x-axis points to the right. The quadrat is a cage-like structure made of dowels spaced every 10 centimeters. It measures 100 cm. in the x and y directions, and 30 cm. in the z direction.

5.2 Getting rectification parameters

5.2.1 Inputs

Because of the dimensions of the quadrat, the near and far faces of the quadrat observed by the side camera have 44 nodes each. The view coordinates are digitized and paired with the face coordinates for each node. The near face data is stored in `grid-sn.txt` and the far face data is in `grid-sf.txt`. Likewise, the near and far faces observed by the top camera have 121 nodes, and are stored in `grid-tn.txt` and `grid-tf.txt`.

5.2.2 Calculation

We calculate the rectification parameters using the following commands.

- `java Hello grid-sn.txt param-sn.txt`
- `java Hello grid-sf.txt param-sf.txt`
- `java Hello grid-tn.txt param-tn.txt`
- `java Hello grid-tf.txt param-tf.txt`

5.2.3 Outputs

At this point the rectification parameters are in the four parameter files with the form `param-xx.txt`. From here on out, we will be working with Excel rather than the command line.

5.3 Rectification

5.3.1 Inputs

We paste the rectification parameters for each camera for each face into a worksheet. Refer to the **CameraParams** worksheet in the example workbook. Compare **CameraParams** to the `param-xx.txt` files.

5.3.2 Calculations

We use the `rectify(xP, yP, x, y)` macro to convert the view coordinates into face coordinates for each camera for each sheet. Refer to the worksheets **Cam1Outputs** and **Cam2Outputs** in the example workbook.

5.3.3 Outputs

The outputs are the rectified 2D tuples in face coordinates.

5.4 Converting 2D face coordinates to 3D world coordinates

We start by taking the rectified 2D coordinates and adding a z-coordinate of 0. Then we transform this coordinate system to conform to the world coordinate system. Because of the way the experiment was designed, these transformations are trivial. However, macros are provided for manipulating a coordinate transformation matrix to provide a much more generalized method. Use of a CTM is demonstrated in the example worksheet **Results**.

5.4.1 Inputs

As described, the input to this step is simply the 2D face coordinates.

5.4.2 Calculations

- Since the far plane viewed by the top camera is defined as $z = 0$ in world coordinates, and the quadrat defines the world coordinate system, all we need to do to convert top-far face coordinates to world is add a z-coordinate of 0. *I.e.* $(faceX, faceY, 0)$. No further transformations are required.
- Since the near plane viewed by the top camera is defined as $z = 30$ in world coordinates, we could similarly add a z-coordinate of 30. To show a more generalized method, we construct a CTM to translate the points 30 cm. in the z direction. The Excel equation to accomplish this is `= translateCTM(identityCTM(), 0, 0, 30)`, which yields the CTM:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Then we use the `transformPoint(ctm, point)` macro to do the transformation.

- Since the near plane viewed by the side camera is defined by $y = 0$ in world coordinates, could do the transformation by converting to $(faceX, 0, faceY)$. Or we could make a CTM to rotate the face coordinate system 90 degrees around the face x-axis, which would do the same thing using `= rotateX(identityCTM(), 90)`, which yields the CTM⁴:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \approx 0 & -1 & 0 \\ 0 & 1 & \approx 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- Since the far plane viewed by the side camera is $y = 100$ in world coordinates, we could just go $(faceX, 100, faceY)$. Or we could make a CTM to rotate the face coordinate system 90 degrees around the x-axis, and then translate it -100 cm. along the z-axis. `= translateCTM(rotateX(identityCTM(), 90), 0, 0, -100)` yields

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \approx 0 & -1 & 100 \\ 0 & 1 & \approx 0 & \approx 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Using the CTMs is confusing, but a very generalized method. If your experimental setup requires it, try out your CTM by converting a few simple trial points like $(1, 2, 0)$ to see that what happens agrees with is you expect to happen. This is the kind of thing I intend the **SCRATCH** worksheet for. For example:

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \approx 0 & -1 & 100 \\ 0 & 1 & \approx 0 & \approx 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \cdot \begin{vmatrix} 1 \\ 2 \\ 0 \\ 1 \end{vmatrix} = \begin{vmatrix} 1 \\ 100 \\ 2 \\ 1 \end{vmatrix}$$

After a while, you'll get better at visualizing the transformations

⁴Due to the limitations of floating point math and the π I used, array elements that should be zero may be shown as very small numbers in Excel

5.4.3 Outputs

The outputs of this step are the four related 3D points where the lines of sight from the camera through the object intersect the near and far planes for each camera.

5.5 Calculating the “intersection” of the lines of sight

5.5.1 Input

The input for this step consists of the four 3D points in world coordinates that represent two lines of sight—one for each camera—passing through the same object at the same time. In this case the object was the head of a fish, and the time was determined by synchronizing the timecode recorded by our cameras.

5.5.2 Calculations

One macro, `threeDLineIntersection(p1, p2, p3, p4)`, returns the 3D point representing the intersection of the two 3D lines.

5.5.3 Outputs

The outputs are a series of 3D points that trace the path of the object in world coordinates.

5.6 Following one set of observations through the calculations

1. The first observation is of the (0, 0, 0) corner of the quadrat. In camera 1 (the side camera) we digitized view coordinates of (150, 142) for this object. In camera 2 (the top camera) we observed the same node at view coordinates of (180, 87). These coordinates are entered in the **ViewData** worksheet.
2. Moving to the **Cam1Outputs** and **Cam2Outputs** worksheet, you can see the following operations moving from view coordinates to face coordinates

$$\text{rectify}(\text{Cam1NearPX}, \text{Cam1NearPY}, \text{ViewData!A6}, \text{ViewData!B6}) = (-0.02471, -0.05217)$$

$$\text{rectify}(\text{Cam1FarPX}, \text{Cam1FarPY}, \text{ViewData!A6}, \text{ViewData!B6}) = (0.52195, -0.18412)$$

$$\text{rectify}(\text{Cam2NearPX}, \text{Cam2NearPY}, \text{ViewData!C6}, \text{ViewData!D6}) = (6.44648, 6.08974)$$

$$\text{rectify}(\text{Cam2FarPX}, \text{Cam2FarPY}, \text{ViewData!C6}, \text{ViewData!D6}) = (0.52195, -0.18411)$$

3. On the same output worksheets, we add a 0 z coordinate to the face coordinates, and then operations like this transform view coordinates to world coordinates

$$\text{transformPoint}(\text{Cam1NearCTM}, \text{E4} : \text{G4}) = (-0.024709243, -8.42907E - 17, -0.052174841)$$

This process yields four 3D points:

(-0.024709243, -8.42907E - 17, -0.052174841)
(-42.93732678, 100, -20.4888376)
(6.446487155, 6.089748471, 30)
(0.521957189, -0.18411516, 0)

4. then, we pass the four points to

threeDLineIntersection(p1, p2, p3, p4)

which gives the result

(0.313977921, -0.273428962, -0.034318177)

It turns out the error is about 0.4 cm., which we can calculate because we know the answer should be (0, 0, 0).

6 Request

I request that if you use this code you acknowledge our contribution to your work. I would also appreciate a note telling me how you used the macros. If you were unable to make head or tails of this mess, I'd like to know that too.

References

- [1] D. Hearn and M. P. Baker. *Computer Graphics*. Prentice Hall, Englewood Cliffs, New Jersey 07632, second edition, 1994, 1986.
- [2] N. F. Hughes and L. H. Kelly. New techniques for 3-d video tracking of fish swimming movements in still or flowing water. *Canadian Journal of Fisheries and Aquatic Sciences*, 53(11):2473–2483, 1996.
- [3] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, New York, 1973.